

第2章

浮動小数点割り算回路の最適化設計

――信頼性を確保しつつ高速な演算回路を構築する視点

小林芳直

ここでは，浮動小数点の割り算回路の設計法について解説する．一度に演算するビット数を2，3，4ビットと増やしていくことで，回路の高速化を図る方法を紹介する．なお，本記事で紹介する各演算回路の設計データは本誌のホームページ(http://www.cqpub.co.jp/dwm/)からダウンロードできる．　（編集部）

ここでは，浮動小数点の割り算回路の高速化設計について，事例を交えながら解説します．浮動小数点の基本的な演算には加減算，積算，割り算がありますが，この中でも割り算は際立って処理に時間がかかります．それは，割り算が基本的に引き算の繰り返しになり，高速化が難しいからです．

本稿では，まず，正確に1ビットずつ計算する基本的な方法を説明し，その後，複数ビット(2，3，4ビット)をまとめて計算することで高速化を図る手法を説明します[注1]．

1 演算精度の概要を見積もる

実際に演算回路を作る前に，精度や処理手順などの概要を見積もっておきます．

注1：浮動小数点の割り算は，本稿で紹介するようなビットごとに計算する方法のほかに，除数の逆数を計算して被除数に積算する方法がある．このときの逆数の計算方法には，区間を分割して4次程度の近似式で解く方法と，ニュートン・ラプソン漸近式で解く方法がある．

● 浮動小数点の構造をおさらい

まず，浮動小数点について簡単におさらいしておきましょう[1]．図1に示すように，単精度浮動小数点数は符号1ビット，指数8ビット，仮数23ビットの合計32ビットで構成されます．符号ビットは，‘0’の場合は正数を，‘1’の場合は負数を表します．指数ビットは正数であり，1～254の値をとります．この値から127(オフセット)を引いた数，つまり－126～＋127で指数eを表します．このとき，小数点より上の数はかならず‘1(暗黙の1)’となり，仮数は小数点以下の数字を23ビットで表すことになります．ですから，仮数の精度としては，小数点の上の‘1’を含めて24ビットとなります．なお，仮数の最下位ビットを「ulp(unit in tha last place)」と言い，浮動小数点の演算精度を表すために用いられます．

ところで，指数は8ビットで構成されるため，0と255という値もとりえます．この場合，仮数の値によって異なった意味を持ちます．

- 指数＝255，仮数≠0：NaN(not a number；非数)．演算エラーが起きたことを意味する．演算エラーは，0で除算したり，＋∞から＋∞を引いた場合に発生する
- 指数＝255，仮数＝0：無限大．これは，演算結果の指数が＋128より大きくなったことを意味する．無限大に

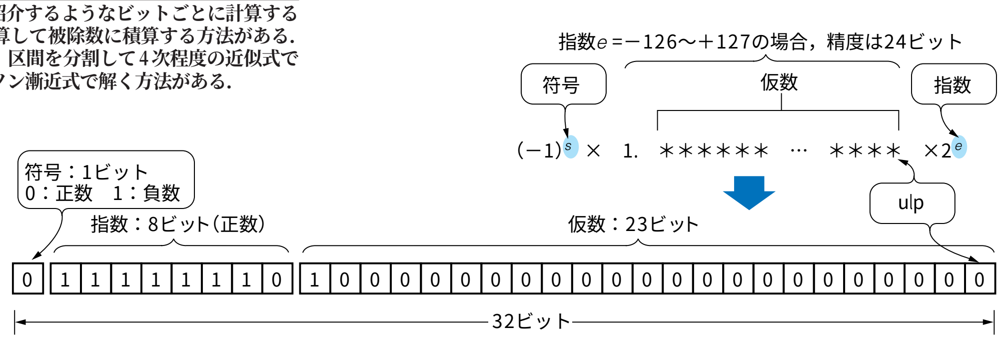
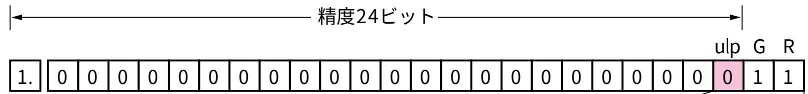


図1
単精度浮動小数点数の構造
単精度浮動小数点数は32ビット幅であり，符号1ビット，指数8ビット，仮数23ビットで数字を表す．

図2
GビットとRビットを使って丸めを表現
Rビットは，Gビットより下位の値をすべてORしたもの．



ulp	Gビット	Rビット	丸め
0	0	0	－
0	0	1	－
0	1	0	－
0	1	1	＋1：切り上げ
1	0	0	－
1	0	1	－
1	1	0	＋1：切り上げ
1	1	1	＋1：切り上げ

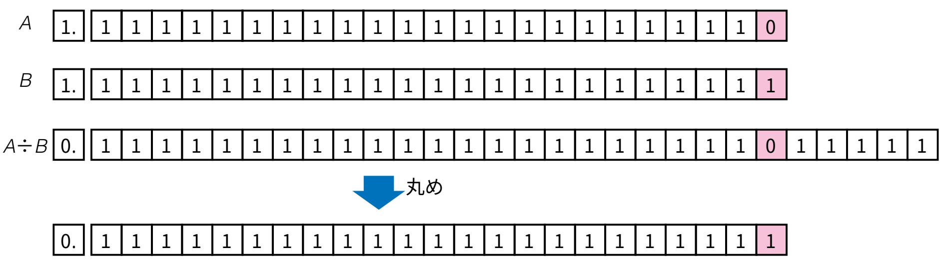


図3　割り算におけるけた上がりの発生
割り算では最初の$A \geqq B$の真偽で先頭ビットの‘1’の位置が確定すると，丸めでその位置が移動するということはない．

は，＋∞と－∞がある

- 指数＝0，仮数＝0：ゼロ．これは演算結果の指数が－127より小さくなったことを意味する
- 指数＝0，仮数≠0：不正規化数となり，小数点より上の数は‘0’になり，指数eは－126で固定される

● 浮動小数点における四捨五入「丸め」

浮動小数点による演算の結果が仮数の精度である24ビットという範囲からあふれた場合，演算の最後に「丸め」という作業が入ります．

どのように丸めを行うかは，**図2**に示すようにulpの下位ビットによって決まります．ここで，ulpの一つ下位のビットをG(guard)ビット，Gビットから下の値をすべてORしたものをR(round)ビットと言います．**図2**の表を論理圧縮すると「G and (ulp or R)」となるため，G＝1のとき，ulpまたはRビットが‘1’であれば切り上げが発生します．それ以外の場合は切り捨てとなります．

ところで，浮動小数点の割り算では，この丸め処理のために演算の途中で数ビット余分な精度で計算することになります．具体的に必要となるビット数は以下のとおりです．

- 25ビット($A \geqq B$判定用1ビット＋24ビット)
- 余りが‘0’かどうかを判定するビット

割り算の場合，被除数と除数の仮数を比較すれば，答えが1以上か1未満かを簡単に判定できます．被除数A，除数Bの変域は仮数についてそれぞれ$1 \leqq A < 2$，$1 \leqq B < 2$なので，計算結果は以下の範囲に収まります．

$$0.5 < A/B < 2 \begin{cases} A \geqq B \text{ならば} 1 \leqq A/B < 2 \\ A < B \text{ならば} 0.5 < A/B < 1 \end{cases}$$

この性質を用いると，答えが1以上か1未満かを判定するには，まず$A \geqq B$を判定し，$A \geqq B$が真であれば最初のビットは‘1’となり，残りの24ビットを計算すればよいとわかります．もし，$A \geqq B$が偽であれば，最初のビットは‘0’，次のビットは‘1’となり，その後の24ビットを計算すればよいのです．

このことから，上記の両方の場合に対応するためには，答えを26ビットで計算して丸めるということになります．

● 除算ではけた上がりに備えて丸め結果を待つ必要はない

丸めた瞬間にけた上がりが起こり，最下位からキャリが伝播して最上位ビットが移動することがあります．計算結果が25ビット以上連続して‘1’となった場合に発生します．けた上がりが起こったものを丸めると‘2’になりますが，このような丸めによるキャリが発生するのであれば，丸めの結果を待たないと指数を確定できません．

こうした現象は積算の場合にしばしば発生しますが，丸めの結果によって小数点の位置が移動してしまい，指数に1を加えるという作業が必要になります．＋1する前の指数が254の場合，先述のNaNになる可能性もあります．

積算の場合は，掛け合わせる二つの数CとDの変域が$1 \leqq C < 2$，$1 \leqq D < 2$なので，答えの範囲は$1 \leqq C \times D < 4$となり，2で周辺のこうした問題が発生します．しかし，積算の場合は答えが2以上になるかどうかを前もって判断する方法がないため，仮数の計算を終了しないと指数は確定しません．しかも，丸めの結果まで待たないと指数は決定できないのです．

では，割り算のけた上がりはどうなるのでしょうか．割り算の答えの変域は$0.5 < A/B < 2$なので，“キャリ事件”

が発生するとすれば，それは1の周辺です．例えば，図3のA，Bのように，Bのほうがわずかに大きい場合はどうでしょう．けた上がりによって演算結果が1になるケースがあるかどうかですが，図からわかるように丸めの結果キャリが発生して最上位けたが移動することはありません．また，この値よりも1に近いものは存在しないので，キャリの伝播によって1未満の数字が1またはそれ以上になることはないとわかります．

つまり，割り算では最初の$A \geqq B$で左端の1の位置が確定すると，丸めでその位置が移動するということはないので，あらかじめ指数まで確定できるのです．

ただし，一つだけ例外があります．被除数　2(＜2)を2で割り，その演算結果の指数が－126の場合，

$$0.11111111111111111111111111 \times 2^{-126}$$
$$= 2^{-126}$$

に丸められます．この演算では丸めの段階で瞬間的に指数が‘0’になってから‘1’に戻っています．割り算においてこのような現象が起こるのは，この場合しかありません．

2 基本的な割り算の演算回路

それでは，具体的な割り算の演算回路の設計に話を進めましょう．

まずは答えを1ビットずつ計算する回路です．この回路は，引き算で1ビットずつ計算するという作業を26回(先ほど演算精度を26ビットとしたため)繰り返してから，余りが0になるかどうかを判定して丸めを行う，というものです．この手順を次のようにまとめました．

1) 被除数Aと除数Bを取り込む
2) $A \geqq B$が真であれば，
 $A \leqq A - B$
 $C \leqq C$&‘1’
 として，4)に進む
3) $A \geqq B$が偽なら
 $C \leqq C$&‘0’
4) Aを1ビット左にシフト
 2)に戻る．ここまでを26回繰り返す
5) Cを正規化する

この回路は，引き算に26クロックを要するほか，外部データを中に取り込むための1クロックと，演算結果を丸めて外に出すための1クロックの合計28クロックが必要です．これをステート・マシンにするときは，アイドル状態を追加して表1のような状態が必要になります．つまり，ステート・マシンは29の状態で構成されます．ベース・カウンタとしてジョンソン・カウンタを選ぶと，15ビットが必要になります．15ビットのジョンソン・カウンタは30状態を表現できるので，どれか一つの状態を捨てることになります(Nビットのジョンソン・カウンタは2Nの状態数を表現できる)．今回は，“000000000000001”を未使用にしました．

表1　1ビットずつの演算に必要な状態

状　態	処理内容
状態0	アイドル状態．CalReqの信号で状態1に移動
状態1	データ読み込み
状態2	1ビット目を計算
状態3	2ビット目を計算
状態4	3ビット目を計算
状態5	4ビット目を計算
⋮	⋮
状態25	24ビット目を計算
状態26	25ビット目を計算
状態27	26ビット目を計算
状態28	丸め

● ブービ・トラップ回路を組み込んで信頼性を上げる

ステート・マシンの誤動作を防ぐためにブービ・トラップ回路を入れることがあります．ブービ・トラップ回路には，ステート・マシンが異常状態になったときに正常状態に復帰させる働きがあり，これは信頼性の高いステート・マシンを設計するために必要な回路テクニックです．

ブービ・トラップ回路があってもなくても，ESD(electro static discharge；静電気放電)やPLD(power line disturbance；電源電圧変動)，雷サージなどによって回路が誤動作することはあります．こうした状況で，ステート・マシンが異常状態(設計段階で未定義だった状態)から正常状態に復帰できれば一過性の不ぐあいですみます．一方，復帰できない場合はシステム・ダウンにつながります．

解決策の一つは，異常状態を検出してシステムにリセットをかけ，正常状態に復帰させる方法です．もう一つは異常状態から正常状態に戻る方法を設計の中に織り込んで，いかなる状態になっても正常状態に戻る機能をステート・マシンの一部にしてしまう方法です．前者は，異常状態の検出に失敗すると正常復帰できなくなりますが，後者は確実に正常復帰することができます．両方を併用することで，

より安全なステート・マシンを設計できます．

ジョンソン・カウンタの場合，ブービ・トラップ回路には定石があります．それは，末尾（下位）ビットが全ビット数の1/3より長く連続して‘0’になった場合，先頭ビットをセット（‘1’）するというものです．15ビットのジョンソン・カウンタであれば，ビット数の1/3は5ビットなので，それより長い6ビットの連続した末尾の‘0’，つまり“---------000000”が先頭ビットのセット条件になります．先頭ビットのリセット条件は，末尾のビットが‘1’となったときとします（**図4**）．

このブービ・トラップが実際に効いているかどうかですが，結論から言うと起こりうるすべてのパターンについて30クロック以内で正常状態に戻ります．これを解析するには，初期状態において先頭ビットが‘1’の場合と‘0’の場合について，連続した6個の‘0’が存在するかしないか（セット条件が成立しうるかどうか）を考えます．例えば，初期状態が“1XXXXXXXXXXXXXX”で，この中に連続した6個の‘0’が含まれていない場合（セット条件が成立しえない場合），右端の6ビットのうちのいずれかが‘1’なので，それが最下位ビットにきたとき（リセット条件が成立したとき）に先頭ビットが‘0’になります．その後はセット条件がないので，15クロック後には正常状態に戻ります（**図5**）．そのほかのパターンについてはここでは説明を割愛します．みなさんで確認してみてください．

● 丸めの効率化を考えた回路設計

先に，浮動小数点の場合，最後に丸めという処理が必要であると述べました．すべての演算結果を求めてから丸めるのではなく，丸めにつごうの良い演算方法を使うことで演算の効率が上がります．

26ビット全部を計算してから丸めると，26ビットから25ビットを切り出す回路が必要になります．丸めの精度のところでも話したように，被除算$A \geqq$除算Bが真であれば，最初のビットは‘1’で，残り24ビットを計算します．$A \geqq B$が偽であれば頭の2ビットが“01”となって残りの24ビットを計算すればよいのですが，この$A \geqq B$の真偽は演算の早い段階で判明しています．

この特徴を生かして，演算結果を格納するレジスタ（結果レジスタ）RegCに必要な25ビットがきちんと整った状態で入るようにすれば，データ・セレクタは不要になります．具体的には，$A \geqq B$が偽のときだけは演算結果を1ビット左にシフトしてから結果レジスタRegCに入れるようにすれば，$A \geqq B$の真偽にかかわらず結果レジスタには必要な25ビットが求められます（**図6**）．このとき，レジスタRegCの先頭ビットはつねに‘1’となるため，レジスタにはこのビットの値を残す必要はありません．そのため，結果レジスタRegCは24ビットですみます（何も考えずに26ビットを計算すると，結果レジスタは26ビット必要）．こうして，整列（アライメント）の効果で周辺回路がわかりやすくなるばかりでなく，回路規模も小さくなります．

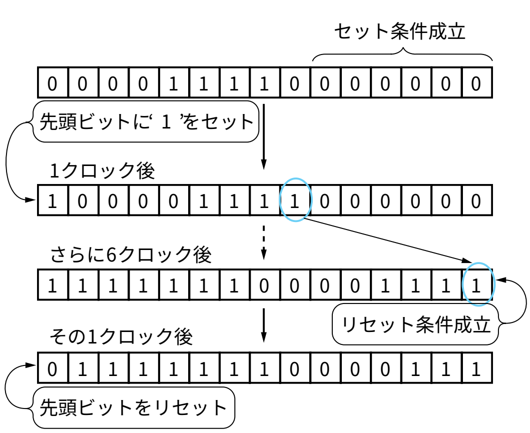


図4　ジョンソン・カウンタのブービ・トラップ

15ビット・ジョンソン・カウンタでは，下位6ビットが連続して‘0’となった場合をセット条件，最下位ビットが‘1’の場合をリセット条件としている．これは，ブービ・トラップ回路の定石．

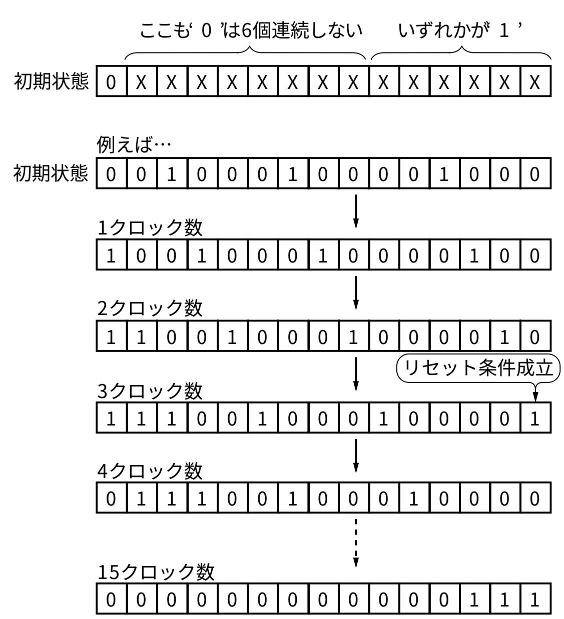


図5　初期状態に連続した6個の‘0’が含まれない場合

ブービ・トラップの確認の例．初期状態に連続した6個の‘0’が含まれていない場合（セット条件が存在しない場合），15クロック後には正常状態（図の例では‘000000000000111’）に戻る．

なお，アライメントを考えると，状態とその動作内容が変わってきます．**表1**では，状態1でデータを読み込んだ後，状態2からすぐに1ビット目を計算しています．これに対して，アライメントを施した場合は，状態1でデータを読み込み，状態2でアライナをセットアップし，状態3で最初の‘1’の位置を合わせます．上述したように，先頭ビットの‘1’を合わせれば後の24ビットを計算すればよいので，これを状態4～27で順に行います．状態28で丸めの作業を行うのは同じです．

● 計算は「指数」→「符号」→「仮数」の順番で

割り算の演算開始後，最初のクロックで被除数A，除数Bの指数を計算します．この計算は次のように行います．

- $A \geqq B$が真のとき：$e_a - e_b + 127$
- $A \geqq B$が偽のとき：$e_a - e_b + 126$

（ここで，e_a，e_bはAとBの指数ビットの値を表す）

前述のように，127のオフセットを加えて指数を得るため，指数ビット＝127において指数は0になります．しかし，引き算を行うとオフセットが打ち消されてしまうので，計算の直後にあらためて加算するわけです．

AとBが正規化数の場合，$e_a - e_b$の最上位ビットを反転して，

- $A \geqq B$が真のとき：$e_a - e_b - 1$
- $A \geqq B$が偽のとき：$e_a - e_b - 2$

としても，先の指数計算と同じ結果が得られます．

A，BにはNaN，無限大，ゼロという状態もあるので，こうした場合の処理も必要です．一度，NaNや無限大，ゼロといった状態になると正規化数には戻れません．また，正規化数を演算した結果，無限大またはゼロになる場合があります．これらを整理すると**表2**のようになります．

演算が始まった最初のクロックで符号を決定します．A，Bの符号が異なる場合は‘1’，同じ場合は‘0’になるので，A，Bの符号ビットのEXOR（排他的論理和）をとればよいことになります．符号の計算はこれで終わりです．

仮数の計算も演算が始まった最初のクロックから開始します．最初のクロックでは次の作業を行います．

- $A \geqq B$が真のとき：計算結果をAlignレジスタに入れる
- $A \geqq B$が偽のとき：Aを左に1ビット・シフトする

次のクロックからは，

表2 NaN，無限大，ゼロの演算処理

AとBの状態		演算結果
A，BのどちらかがNaN		NaN
Aが無限大	Bが無限大	NaN
	Bが正規化数	無限大
	Bがゼロ	NaN
Aが正規化数	Bが無限大	ゼロ
	Bが正規化数	計算実行
	Bがゼロ	NaN
Aがゼロ	Bが無限大	ゼロ
	Bが正規化数	ゼロ
	Bがゼロ	NaN

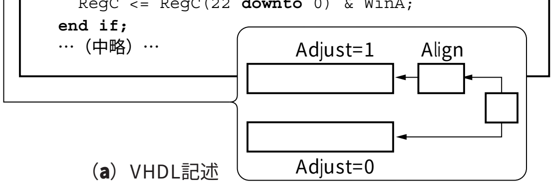
（FDiv1.vhdより一部抜粋）

```
WinA <= '1' when RegA >= ("01" & RegB)
                        else '0';
・・・（中略）・・・
 if WinA = '1' then
    Adjust <= '1';
  else
    Adjust <= '0';
  end if;
end if;
…（中略）…
---------------------------------------------
-- Divider Calculation
---------------------------------------------
Align <= WinA;

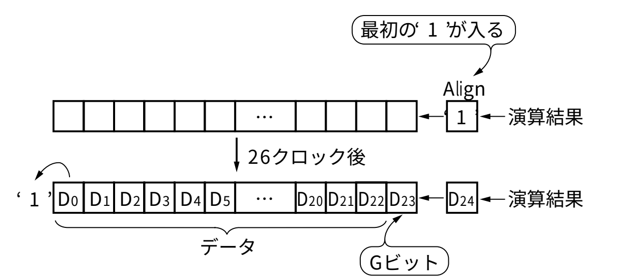
if Adjust = '1' then
  RegC <= RegC(22 downto 0) & Align;
else
  RegC <= RegC(22 downto 0) & WinA;
end if;
…（中略）…
```

A≧Bが真ならWinA=1

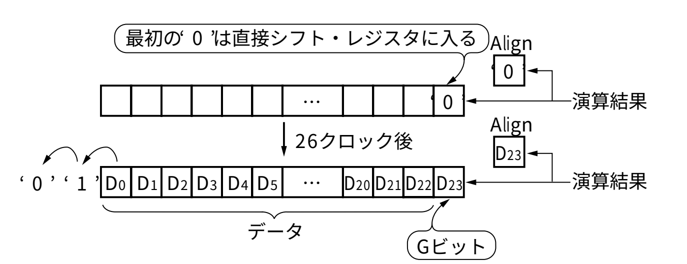
A≧Bが真ならAdjust=1
A≧Bが偽ならAdjust=0



（a）VHDL記述



（b）アライナの動き（Adjust＝1）



（c）アライナの動き（Adjust＝0）

図6 アライメント

結果レジスタに演算結果をきちんと整列して入れれば，26ビットから必要な25ビットを取り出せる．$A \geqq B$が真のときは演算結果をAlignレジスタに入れ，$A \geqq B$が偽のときは演算結果を左に1ビット・シフトしてから結果レジスタに格納する．

表3 オーバラップを取り入れた場合のステート・マシン

状 態	処理内容
状態0	アイドル状態
状態1	次にRegAとRegBを使用
状態2	RegAとRegBを使用，次にRegCを使用
状態3 ⋮ 状態27	RegA，RegB，RegCを使用
状態28	RegCを使用

- $A \geqq B$が真のとき：$A-B$を2倍してAに入れる
- $A \geqq B$が偽のとき：Aを2倍してAに入れる

という処理を繰り返します．

● レジスタ構造を理解してカウンタの重ね合わせを行う

この割り算回路を連続で動作させるときは回路の起動を少し早くすることができます．演算結果が確定する状態28の後，状態0には進まず，状態1に進んでかまいません．

もう少し突っ込んで考えると，状態28と状態1は重なっていてもかまいません．状態28と状態1は内部レジスタを使う状態なのです．回路の内部構造まで知っていると，状態28が使うのはRegCだけであり，RegAとRegBはすでに開放されていることがわかります．また，状態1が次に使うのはRegA，RegBだけです．このようなことから，状態28は状態1と重なっていてもかまわないと言えるのです．

表3に，このステート・マシンの動作を整理してみました．このステート・マシンの起動は，状態26，27，28，0の4ヵ所から行えます．厳密に言うと，このステート・マシンの未使用状態である状態29からでもスタートできますが，状態28と状態29は縮退させているので回路には現れません．かくして，状態ビット0のセット条件は「状態ビット9～14の6ビットがすべて‘0’」ではなく，「状態ビット8～10がすべて‘0’」になります．ただし，リセット条件は状態ビット14だけが‘1’ではなく，「状態ビット11～14のすべてが‘1’」となります．このようにジョンソン・カウンタを少しだけ重ねて使うことを「オーバラップ」と言います．TAT(turn around time)は上がりませんが，スループットを向上させることができます．

● 遊び心の中から，使える回路アイデアが生まれる

回路を実現するにはさまざまな方式があります．すでに紹介した結果レジスタのアライメント($A \geqq B$が偽のとき演算結果を2倍する)のほか，演算の最初だけを2ビット単位で計算する方法[注2]などがあります．

前者の場合，3クロック目(状態3)の処理は先頭ビットの‘1’を合わせることとわかっているので，この処理は省いてもかまいません．しかし，省いたところで，別段，動作速度が速くなるわけでもなく，回路が簡単になるわけでもありません．つまり，これは「遊びの回路設計」ということになります．

ところが，このタイミングにおいて，結果を入れるためのRegCの動作を止めることは効果があります．これはRegCの動作条件を，「DivCyc1 or DivCyc12」から「DivCyc2 or DivCyc12」に変更するだけなので回路規模は変わりませんが，多少消費電力が下がります．

結果レジスタの処理についても演算の最初に0クリアしたほうが気持ち良いのですが，演算の最後にはかならず不定ビットが追い出されているので，0クリアの結果は演算結果に表れません．つまり，0クリアはむだな処理ということになります．

演算の最初だけ2ビット単位で計算する方法では，最初に2ビット計算できる回路を常時動作させれば，25％の確率でこの回路を動かすことができます．そのため，CPI(clock per instruction)を平均25％低減できます．しかし，処理クロック数がばらつく回路は外部回路との接続が難しくなるので，かえって使いにくくなる可能性もあります．どうせ速くするなら，つねに2倍の速度で演算を進められる回路を書いてしまえばよいのです．これが次にお話しする回路設計例になります．

回路を設計するときはいろいろな定石やアイデアが湧き上がってきますが，そのすべてを採用するのではなくて，10の妄想(遊び心)と1のアイデアぐらいに考えて，最適な設計をしたいものです．

3 2ビットずつ計算する割り算回路

次は，2ビットずつ計算していく割り算回路です．この回路の処理手順を**図7**に示します．この図のように引き算器を三つ用意することで，2ビットずつの演算を行えます．答えは25ビット求める必要があり，これに13クロックを要します．また，データのロードと出力に1クロックずつ使うので，合計15クロック必要になります．

注2：この方法では，$A \geqq B$の真偽に応じて‘1’あるいは‘01’を結果レジスタに入れる．バレル・シフタは不要だが引き算回路が2系統必要になり，周辺回路が複雑になる．ただし，ほかの方法と比べて処理時間が1クロック少なくなる．

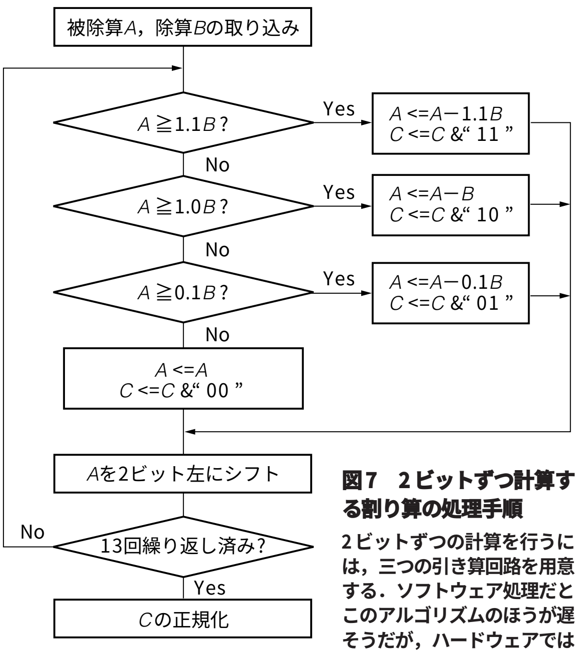


図7　2ビットずつ計算する割り算の処理手順
2ビットずつの計算を行うには，三つの引き算回路を用意する．ソフトウェア処理だとこのアルゴリズムのほうが遅そうだが，ハードウェアでは2倍の処理速度が出せる．

表4　2ビットずつの演算に必要な状態

状　態	処理内容
状態0	アイドル状態．CalReqの信号で状態1に移動
状態1	データ読み込み
状態2	先頭ビットの‘1’を整列する
状態3	1，2ビット目を計算
状態4	3，4ビット目を計算
状態5	5，6ビット目を計算
状態6	7，8ビット目を計算
⋮	⋮
状態13	21，22ビット目を計算
状態14	23，24ビット目を計算
状態15	丸め

リスト1　三つの引き算回路

```
… (中略) …
WinA3 <= ‘1’ when (RegA
                   & ‘0’)
      >= ParmB3 else ‘0’;
WinA2 <= ‘1’ when RegA
       >= (“01” & RegB )
                 else ‘0’;
WinA1 <= ‘1’ when (RegA
                   & ‘0’)
>= (“001” & RegB )
                 else ‘0’;

ParmB3 <= (“01” & RegB &
                     ‘0’)
      + (“001” & RegB );

SubData3 <= (RegA & ‘0’)
               - ParmB3;
SubData2 <=  RegA
       - (“01” & RegB );
SubData1 <= (RegA & ‘0’)
      - (“001” & RegB );
```

この回路でも最初の1ビット目だけは特別扱いして，先頭ビットをそろえるようにします．ステート・マシンの状態と処理は**表4**のような関係になります．このステート・マシンは全部で16状態あるので，8ビットのジョンソン・カウンタを使うことにします．

また，1ビットずつ計算する場合と同じように考えると，連続で動作させるときは状態1と状態15を重ね合わせることができます．こうすることで，TATは16クロック，スループットは13クロックになります．

● 引き算回路は三つ用意する

この回路では，まずBの3倍，2倍，1倍の値を作り，これとAを2倍した値を比較します．比較結果はWinA3，WinA2，WinA1の三つの信号で表します（**リスト1**）．

WinA3：$A \geqq 1.1B$
WinA2：$A \geqq B$
WinA1：$A \geqq 0.1B$

バイナリの1.1は10進数の1.5を，バイナリの1.0は10進数の1.0を，バイナリの0.1は10進数の0.5を表す．

これに対応して，引き算回路は次の三つを用意します．

SubData3 $= A - 1.1B$
SubData2 $= A - B$
SubData1 $= A - 0.1B$

WinA3が成立したときの答えは“11”になります．そのときの$A - 1.1B$の結果の先頭2ビットは“00”になります．これを2ビット左にシフトしてRegAに入れます．WinA2が成立したときの答えは“10”になります．$A - B$の結果の先頭2ビットは“00”になり，これを2ビット左にシフトしてRegAに入れます．WinA1が成立したときの答えは“01”になります．このとき$A - 0.1B$を行うと結果の先頭2ビットは“00”になり，これを2ビット左にシフトしてRegAに入れます．

このようにして2ビットずつ計算を進めていきます．

● アライナかレベル・シフタか

1ビットずつ計算する割り算回路の場合と同じく，$A \geqq B$の真偽によって結果レジスタへのビットの入れかたを変えると（つまりアライメントを行うと），丸めが簡単になります．具体的には，Alignという1クロック遅れのラッチに計算データを入れて，以下のような処理を行うことで，最上位の‘1’の位置をそろえることができます（**図8**）．

● $A \geqq B$が真のとき

```
Align <= CalData(0)
結果レジスタ <= Align(1回遅れのCalData(0))
                                & 新しいCalData(1)
```

● $A \geqq B$が偽のとき

```
結果レジスタ <= 新しいCalData(1 downto 0)
```

先頭ビットの位置が確定しているのであれば，そのビットは見る必要がないので，結果レジスタは24ビットで済む

（FDiv2.vhdより一部抜粋）

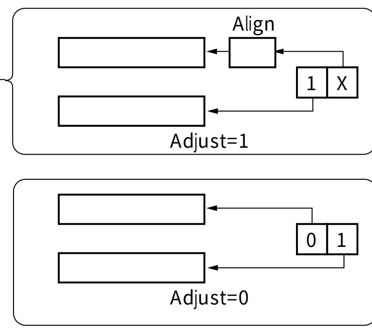
```
…（中略）…
 if DivCyc1 ='1' and DivCyc2 ='0' then
  if WinA2 ='1' then
    Adjust <='1';
  else
    Adjust <='0';
  end if;
end if;
------------------------------------
-- Mantissa Calculation
------------------------------------
if DivCyc0 ='1' and DivCyc1 ='0' then
  RegA <= "01" & FloatA(22 downto 0);
else
  if WinA3 ='1' then
    RegA <= SubData3(23 downto 0) &'0';
  elsif WinA2 ='1' then
    RegA <= SubData2(22 downto 0) &"00";
  elsif WinA1 ='1' then
    RegA <= SubData1(23 downto 0) &'0';
  else
    RegA <= RegA(22 downto 0) &"00";
  end if;
end if;

if DivCyc0 ='1' and DivCyc1 ='0' then
  RegB <= FloatB(22 downto 0);
end if;
```

A≧Bが真なら Adjust=1
A≧Bが偽なら Adjust=0

```
------------------------------------
-- Divider Calculation
------------------------------------
Align <= CalData(0);

if Adjust ='1' then
  RegC <= RegC(21 downto 0) & (Align
                          & CalData(1));
else
  RegC <= RegC(21 downto 0) & CalData;
end if;
…（中略）…
```



◀ **図8　2ビットずつの割り算回路のアライメント**
Alignという1クロック遅れのラッチを用いて，最上位の'1'の位置をそろえる．

表5　3ビットずつの演算に必要な状態

状　態	処理内容
状態0	アイドル状態．CalReqの信号で状態1に移動
状態1	データ読み込み
状態2	先頭ビットの'1'を整列する
状態3	1，2，3ビット目を計算
状態4	4，5，6ビット目を計算
状態5	7，8，9ビット目を計算
⋮	⋮
状態9	19，20，21ビット目を計算
状態10	22，23，24ビット目を計算
状態11	丸め

▶ **リスト2**
比較のための引き算回路

```
…（中略）…
WinA7 <= '1' when (RegA & "00")  >= ParmB7 else '0';
WinA6 <= '1' when (RegA & '0')   >= ParmB3 else '0';
WinA5 <= '1' when (RegA & "00")  >= ParmB5 else '0';
WinA4 <= '1' when RegA           >= ("01" & RegB )
                                        else '0';
WinA3 <= '1' when (RegA & "00")  >= ( '0' & ParmB3)
                                        else '0';
WinA2 <= '1' when RegA & '0'     >= ("001" & RegB )
                                        else '0';
WinA1 <= '1' when (RegA & "00")  >= ("0001" & RegB )
                                        else '0';

ParmB7 <= ("01" & RegB & "00") + ("001" & RegB & '0') +
                                              ("0001" & RegB );
ParmB5 <= ("01" & RegB & "00") + ("0001" & RegB );
ParmB3 <= ("01" & RegB & '0')  + ("001" & RegB );

SubData7 <= (RegA & "00") - ParmB7;
SubData6 <= (RegA & '0')  - ParmB3;
SubData5 <= (RegA & "00") - ParmB5;
SubData4 <=  RegA         - ("01" & RegB );
SubData3 <= (RegA & "00") - ( '0' & ParmB3);
SubData2 <= (RegA & '0')  - ("001" & RegB );
SubData1 <= (RegA & "00") - ("0001" & RegB );
…（中略）…
```

【信号】
WinA7：$A≧1.11B$
WinA6：$A≧1.10B$
WinA5：$A≧1.01B$
WinA4：$A≧B$
WinA3：$A≧0.11B$
WinA2：$A≧0.10B$
WinA1：$A≧0.01B$

【引き算回路】
SubData7＝$A－1.11B$
SubData6＝$A－1.10B$
SubData5＝$A－1.01B$
SubData4＝$A－B$
SubData3＝$A－0.11B$
SubData2＝$A－0.10B$
SubData1＝$A－0.01B$

ことになります．これに対して，演算回路から出てくる2ビットを単純に組み上げていくだけでは，結果レジスタの先頭の'1'の位置も確定できません．そのため，結果レジスタは26ビット必要になり，計算が終わったところであらためて先頭の'1'のビットを見つけ，整列して丸める必要があります．

両者を比較すると，最初にビットを整列してから結果レジスタに送ると1ビットのアライナが必要になりますが，結果レジスタは24ビットですみます．一方，結果を求めてから整列する場合は，先頭の'1'を見つける回路と2→1ビットのバレル・シフタが必要になります．

このようにアライナを先頭に置いておくほうが，回路も処理も簡単になります．

● メインの回路は最初はシンプルに設計する

アライメントを行うだけなら，結果レジスタへのデータの取り込みを1ビットと2ビットで切り替えても実現できます．まず，$A≧B$が真の場合，先頭ビットが'1'になる

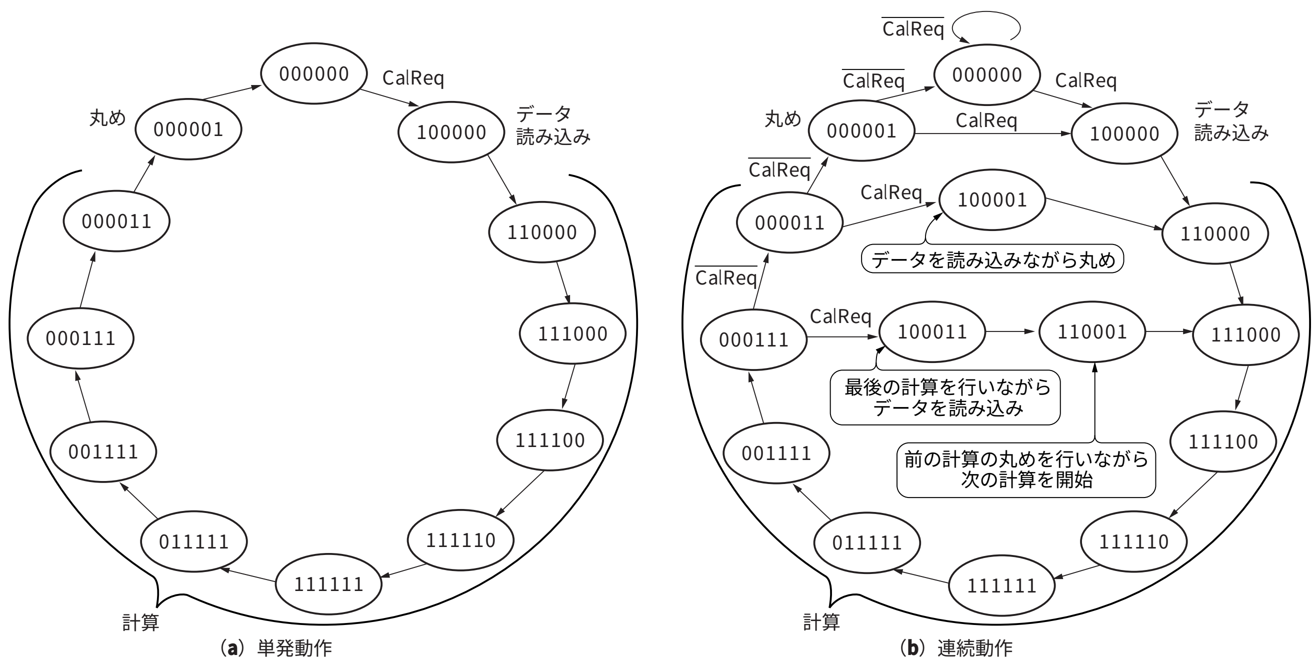


図9　3ビットずつ計算する割り算回路のステート・マシン

ので，RegAには$A-B$の結果を1ビット左にシフトしてから入れます．一方，$A \geqq B$が偽の場合，$A-0.1B$を2ビット左にシフトしてから入れます．

これを実現するためには，RegAに入れるデータとして以下の5種類が必要となります．

- $A-1.1B$の結果を2ビット左シフトした値
- $A-B$の結果を2ビット左シフトした値
- $A-1.1B$の結果を1ビット左シフトした値
- $A-0.1B$の結果を2ビット左シフトした値
- RegAの結果を2ビット左シフトした値

また，結果レジスタには1ビット・シフトと2ビット・シフトを切り替えるしくみが必要になります．

これに対して，RegCの入り口にアライナを入れた場合は次のデータが必要となります．

- $A-1.1B$の結果を2ビット左シフトした値
- $A-B$の結果を2ビット左シフトした値
- $A-0.1B$の結果を2ビット左シフトした値
- RegAの結果を2ビット左シフトした値

つまり，結果レジスタはつねに2ビット・シフトを行えばよく，RegAの周辺回路も簡単になります．

同じ機能を実現する方法がいくつかあるとき，本体はできるだけシンプルな機能にして，外部に小技をきかせて作ったほうが回路は簡単になります．しかし，回路は少し読みにくくなるので，まずは回路規模が大きくても読みやすい回路を作っておき，正しく動作することを確認してから回路規模を削っていく作業を入れるとよいでしょう．

4　3ビットずつ計算する割り算回路

次は，3ビットずつ計算していく割り算回路です．この場合，引き算器を七つ用意してやれば，3ビットずつの演算が行えます．答えは25ビット求める必要があるので，これに9クロックを要します．また，データのロードと出力に1クロックずつ使うので，合計12クロックになります．

これまでの回路と同じように，ここでも先頭ビットをそろえるようにします．ステート・マシンの状態と処理の関係を**表5**に示します．このステート・マシンは全部で12状態あるので，6ビットのジョンソン・カウンタを使うことにします．また，連続で動作させるときは，状態1と状態11を重ね合わせることができるので，TATは12クロック，スループットは9 クロックになります．

この回路では，Bの7倍，6倍，5倍，4倍，3倍，2倍，1倍の値とAを4倍した値を比較します．比較結果はWinA7～WinA1の七つの信号で示します．また，これに対応して七つの引き算回路を用意します（**リスト2**）．

WinA7が成立したときの答えは“111”になります．$A-$

遷移前	遷移後	状態	遷移前	遷移後	状態
000000 →	100000	○	100000 →	110000	○
000001 →	100000	→○	100001 →	110000	○
000010 →	100001	→○	100010 →	110001	→○
000011 →	100001	→○	100011 →	110001	○
000100 →	100010	→○	100100 →	110010	→→○
000101 →	100010	→○	100101 →	110010	→→○
000110 →	100011	→○	100110 →	110011	→→○
000111 →	100011	○	100111 →	110011	→→→→○
001000 →	000100	→→○	101000 →	110100	→→→→○
001001 →	000100	→→○	101001 →	110100	→→→→○
001010 →	000101	→→○	101010 →	110101	→→→→○
001011 →	000101	→→○	101011 →	110101	→→→→○
001100 →	000110	→→○	101100 →	110110	→→→→○
001101 →	000110	→→○	101101 →	110110	→→→→○
001110 →	000111	→○	101110 →	110111	→→→→→○
001111 →	000111	○	101111 →	110111	→→→→○
010000 →	001000	→→→○	110000 →	111000	○
010001 →	001000	→→→○	110001 →	111000	○
010010 →	001001	→→→○	110010 →	111001	→○
010011 →	001001	→→→○	110011 →	111001	→○
010100 →	001010	→→→○	110100 →	111010	→→→○
010101 →	001010	→→→○	110101 →	111010	→→→○
010110 →	001011	→→→○	110110 →	111011	→→→○
010111 →	001011	→→→○	110111 →	011011	→→→→○
011000 →	001100	→→→○	111000 →	111100	○
011001 →	001100	→→→○	111001 →	111100	○
011010 →	001101	→→→○	111010 →	111101	→→○
011011 →	001101	→→→○	111011 →	111101	→→○
011100 →	001110	→→○	111100 →	111110	○
011101 →	001110	→→○	111101 →	111110	→○
011110 →	001111	○	111110 →	111111	○
011111 →	001111	○	111111 →	011111	○

図10　オーバラップ時のブービ・トラップ

セット条件を“ -00--- ”，リセット条件を“ ---111 ”とした場合の動作確認．最大5クロックで正常状態に戻っていることがわかる．

1.11*B* の演算結果の上位3ビットは“ 000 ”になるので，これを3ビット左にシフトしてRegA に入れます．WinA6が成立したときの答えは“ 110 ”になります．このときは*A* − 1.10*B* の演算結果の上位3ビットは“ 000 ”になり，これを3ビット左にシフトしてからRegA に入れます．以下，同じ要領でWinA6～WinA1が成立した場合を考えます．くどくなるので，以降の説明は割愛します．

● オーバラップした場合のブービ・トラップ回路

ここで紹介する3ビットずつ計算を行う割り算回路(Fdiv3.vhd)を制御するステート・マシンの動作は，単発動作の場合なら**図9(a)**のように，また連続動作の場合は**図9(b)**のようになります．

図9(b)からもわかるように，連続動作の場合は，演算開始シーケンスと終了シーケンスが重なります．これを実現するには，ジョンソン・カウンタのセット条件とリセット条件が先に説明した定石とは異なった形をとります．まず，定石では6ビットの1/3より長い末尾の連続する‘ 0 ’，つまり“ ---000 ”がセット条件となりますが，ここでは**図9(b)**からセット条件は“ --00-- ”または“ -00--- ”となります．また，“ 000111 ”→“ 10011 ”というように遷移するためには，リセット条件が“ -----1 ”では不つごうだとわかります．そこで，末尾の連続した“ 111 ”をリセット条件とすると動作してくれそうです．

では，実際に**図10**を見ながら確認してみましょう．**図10**は，6ビットのジョンソン・カウンタについて，セット条件を“ -00--- ”，リセット条件を“ ---111 ”とした場合の回路の動作を調べるものです．○は正常状態にあることを示します．また，→○はあと何クロック(→の数がクロック数を表す)で正常状態に復帰できるかを示しています．すべてのパターンにおいて，最大5クロック後には正常状態に戻っています．つまり，セット条件“ -00--- ”，リセット条件“ ---111 ”でブービ・トラップがかかっていることがわかります．

また，この回路でもこれまでと同じようにアライナを入れます．計算結果の3ビットは`CalData(2 downto 0)`とし，**図11**のような処理を行います．このアライナによって，結果レジスタには計算結果の24ビットが整列して入ります．

5　4ビットずつ計算する割り算回路

最後に，4ビットずつ計算していく割り算回路を紹介します．この場合は，15個の引き算回路を用意することで，4ビットずつの計算を実現できます．25ビットの演算結果を求めるのに7クロック，データのロードと出力に1クロックずつで，合計9クロックを要します．連続動作の場合は，状態1と状態9を重ね合わせることで7クロックで動きます．

この割り算回路の状態と処理内容の関係を**表6**に示します．ステート・マシンは全部で10状態あるので，ここでは5ビットのジョンソン・カウンタを用います．

リスト3の回路では，まず*B*の15倍～1倍を作り，これを*A*を8倍した値と比較します．比較結果はWinAF～WinA1の信号で示します．この信号に対応した15種類の引き算回路を用意します．各信号が成立したときに対応す

（FDiv3.vhdより一部抜粋）

```
…（中略）…
  if WinA4 =‘1’ then
    Adjust <= ‘1’;
  else
    Adjust <= ‘0’;
  end if;
end if;

…（中略）…
-----------------------------------------------------------------
-- Divider Calculation
-----------------------------------------------------------------
if  (DivCyc1 =‘1’ and DivCyc2 =‘0’ and WinA4 =‘1’)
  or((DivCyc2 =‘1’ or DivCyc3 =‘1’) and Adjust =‘1’) then
  Align0 <= CalData(1);
else
  Align0 <= CalData(0);
end if;

Align1 <= CalData(0);

if Adjust =‘1’ then
  RegC <= RegC(20 downto 0)
          & Align0 & Align1 & CalData(2);
else
  RegC <= RegC(20 downto 0) & Align0
          & CalData(2 downto 1);
end if;
…（中略）…
```

A≧Bが真ならAdjust=1
A≧Bが偽ならAdjust=0

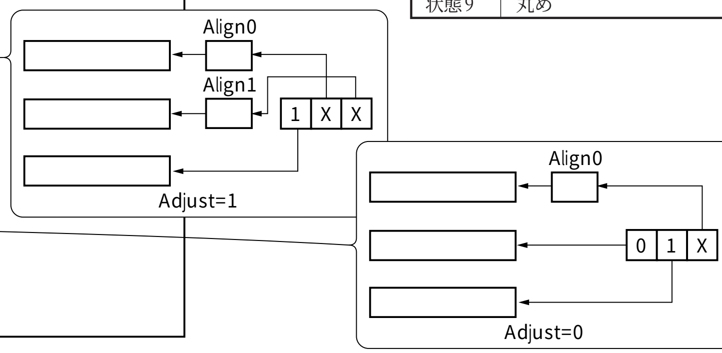


図11　3ビットずつの割り算回路のアライメント
演算結果の3ビットはCalData(2 downto 0)としている．アライメントにより，24ビットのデータが結果レジスタに整列して入る．

▶表6　4ビットずつ計算する演算に必要な状態

状　態	処理内容
状態0	アイドル状態．CalReqの信号で状態1に移動
状態1	データ読み込み
状態2	先頭ビットの‘1’を整列する
状態3	1，2，3，4ビット目を計算
状態4	5，6，7，8ビット目を計算
状態5	9，10，11，12ビット目を計算
状態6	13，14，15，16ビット目を計算
状態7	17，18，19，20ビット目を計算
状態8	21，22，23，24ビット目を計算
状態9	丸め

リスト3　15個の引き算回路

```
…（中略）…
WinAF <= ‘1’ when (RegA & “000”)  >= ParmBF else ‘0’;
WinAE <= ‘1’ when (RegA & “00”)   >= ParmB7 else ‘0’;
WinAD <= ‘1’ when (RegA & “000”)  >= ParmBD else ‘0’;
WinAC <= ‘1’ when (RegA & “0”)    >= ParmB3 else ‘0’;
WinAB <= ‘1’ when (RegA & “000”)  >= ParmBB else ‘0’;
WinAA <= ‘1’ when (RegA & “00”)   >= ParmB5 else ‘0’;
WinA9 <= ‘1’ when (RegA & “000”)  >= ParmB9 else ‘0’;
WinA8 <= ‘1’ when  RegA           >= (“01” & RegB)      else ‘0’;
WinA7 <= ‘1’ when (RegA & “000”)  >= (‘0’ & ParmB7)     else ‘0’;
WinA6 <= ‘1’ when (RegA & “00”)   >= (‘0’ & ParmB3)     else ‘0’;
WinA5 <= ‘1’ when (RegA & “000”)  >= (‘0’ & ParmB5)     else ‘0’;
WinA4 <= ‘1’ when (RegA & ‘0’)    >= (“001” & RegB )    else ‘0’;
WinA3 <= ‘1’ when (RegA & “000”)  >= (“00” & ParmB3)    else ‘0’;
WinA2 <= ‘1’ when (RegA & “00”)   >= (“0001” & RegB )   else ‘0’;
WinA1 <= ‘1’ when (RegA & “000”)  >= (“00001” & RegB )  else ‘0’;

…（中略）…

SubDataF <= (RegA & “000”)   - ParmBF;
SubDataE <= (RegA & “00”)    - ParmB7;
SubDataD <= (RegA & “000”)   - ParmBD;
SubDataC <= (RegA & ‘0’)     - ParmB3;
SubDataB <= (RegA & “000”)   - ParmBB;
SubDataA <= (RegA & “00”)    - ParmB5;
SubData9 <= (RegA & “000”)   - ParmB9;
SubData8 <=  RegA            - (“01” & RegB );
SubData7 <= (RegA & “000”)   - (‘0’ & ParmB7);
SubData6 <= (RegA & “00”)    - (‘0’ & ParmB3);
SubData5 <= (RegA & “000”)   - (‘0’ & ParmB5);
SubData4 <= (RegA & ‘0’)     - (“001” & RegB );
SubData3 <= (RegA & “000”)   - (“00” & ParmB3);
SubData2 <= (RegA & “00”)    - (“0001” & RegB );
SubData1 <= (RegA & “000”)   - (“00001” & RegB );
```

【信号】
WinAF：A≧1.111B
WinAE：A≧1.110B
WinAD：A≧1.101B
WinAC：A≧1.100B
WinAB：A≧1.011B
WinAA：A≧1.010B
WinA9：A≧1.001B
WinA8：A≧B
WinA7：A≧0.111B
WinA6：A≧0.110B
WinA5：A≧0.101B
WinA4：A≧0.100B
WinA3：A≧0.011B
WinA2：A≧0.010B
WinA1：A≧0.001B

【引き算回路】
SubDataF＝A－1.111B
SubDataE＝A－1.110B
SubDataD＝A－1.101B
SubDataC＝A－1.100B
SubDataB＝A－1.011B
SubDataA＝A－1.010B
SubData9＝A－1.001B
SubData8＝A－B
SubData7＝A－0.111B
SubData6＝A－0.110B
SubData5＝A－0.101B
SubData4＝A－0.100B
SubData3＝A－0.011B
SubData2＝A－0.010B
SubData1＝A－0.001B

る引き算を行うと，その結果の先頭4ビットは“0000”となり，これを4ビット左にシフトしてRegAに入れます．考えかたはこれまで説明したほかの割り算回路と同様です．

4ビットずつ計算を行う割り算回路（Fdiv4.vhd）もFdiv.3vhdと同様に，連続動作を行う場合は演算の開始シーケンスと終了シーケンスが重なります．この連続動作を実現するためのジョンソン・カウンタのセット条件は“-0--”，リセット条件は“--111”となります．動作確認はここでは割

（FDiv4.vhdより一部抜粋）

```
… (中略) …
  if WinA8 =‘1’ then
    Adjust <= ‘1’;
  else
    Adjust <= ‘0’;
  end if;
end if;

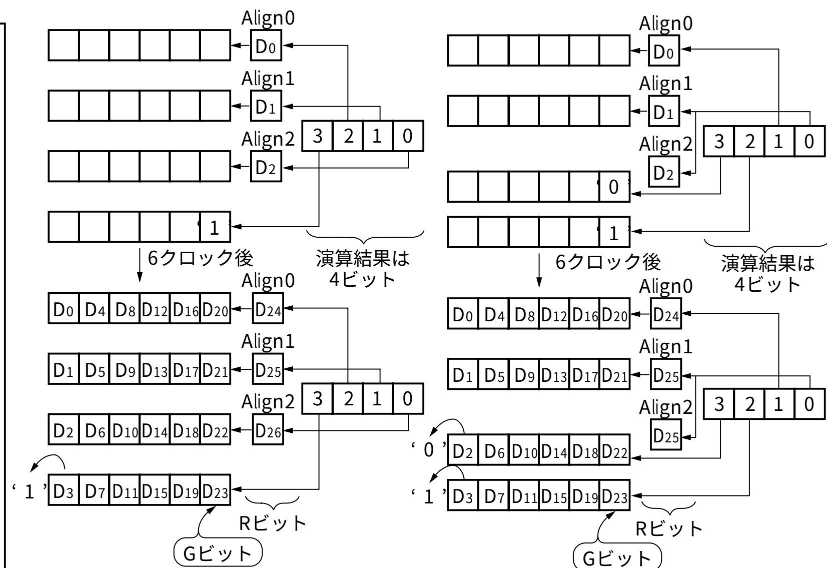
… (中略) …
if  (DivCyc2 =‘0’ and WinA8 =‘1’)
  or(DivCyc2 =‘1’ and Adjust =‘1’) then
  Align0 <= CalData(2);
  Align1 <= CalData(1);
else
  Align0 <= CalData(1);
  Align1 <= CalData(0);
end if;

Align2 <= CalData(0);

if Adjust =‘1’ then
  RegC <= RegC(19 downto 0) & Align0
          & Align1 & Align2 & CalData(3);
else
  RegC <= RegC(19 downto 0) & Align0
          & Align1 & CalData(3 downto 2);
end if;
… (中略) …
```

$A \geqq B$が真なら Adjust=1
$A \geqq B$が偽なら Adjust=0

（**a**）VHDL記述



（**b**）アライナの動き（Adjust=1）　（**c**）アライナの動き（Adjust=0）

図12　4ビットずつの割り算回路のアライメント
(b)と(c)からわかるように，アライナが働いてシフト・レジスタの内容はAdjustの値にかかわらず同じになる．

愛しますが，すべてのパターンにおいて最大4クロックで正常状態に戻ります．

アライナの動作については，**図12**を参照してください．ここでは，計算結果の4ビットはCalData(3 downto 0)としています．

● 論理圧縮は十分に，冗長なタイミングは作らずに

本稿では，4種類の割り算回路を紹介しました．機能が同じでも回路の組みかたによって処理速度が変わることを理解していただけたかと思います．また，回路の細部にはいろいろな選択の余地があり，そこを調整することで回路規模が大きく変わります．

回路が簡単になると思って選択した回路方式なのに，実装してみると配線量が大きくなり，かえって動作速度が落ちてしまうこともあります．回路と配線を比較するとFPGAの場合は配線のほうが負荷は大きくなります．最新のASICでも配線量がチップ・サイズや動作速度を決めてしまいます．今回の設計例でも，一見コンパクトに見えるFdiv4.vhd（4ビット計算の割り算回路）の配線量がもっとも大きくなり，動作速度が遅くなっています．

連続動作を許すかどうかで，スループットも変わります．ステート・マシンの設計そのものも変わります．このあたりで，回路設計のだいご味をお楽しみください．

最後にたいせつなことをひと言．回路設計において十分に論理圧縮が行われていないとか，冗長なタイミングがあるということは，新聞の誤字脱字に相当する事件です．英文のスペックならスペル・ミスがあるとそれだけで読む気が失せるという話があります．回路設計においても基本的な定石はきちんと守っていきましょう．論理圧縮は十分行い，冗長なタイミングは作らず，かつ遊び心も忘れずに設計したいものです．

参考・引用*文献

(1) 小林芳直；浮動小数点演算回路を極める，Design Wave Magazine，pp.45-56，2003年7月号．

こばやし・よしなお
(株)京セラ　ディスプレイ研究所

<筆者プロフィール>
小林芳直．京セラ　ディスプレイ研究所技術開発部長．ここ数年は有機LEDのドライバ，PIXEL回路の設計を行っている．東京大学理学部 情報科学科非常勤講師．第一級陸上無線技術士．愛車はプリウス．趣味はオーディオ．